GigaVoxels, librairie et kit de développement sur GPU pour l'exploration temps-réel et visuellement réaliste d'immenses scènes détaillées à base de SVO

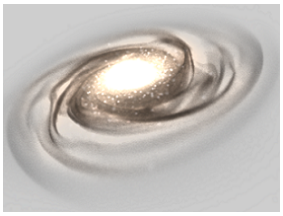
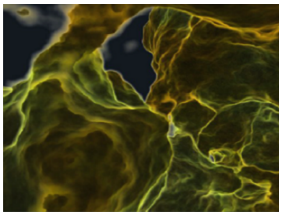
Cyril Crassin, Pascal Guehl, Eric Heitz, Jérémy Jaussaud, Fabrice Neyret

GigaVoxels, librairie et kit de développement sur GPU pour l'exploration temps-réel et visuellement réaliste d'immenses scènes détaillées à base de SVO

Cyril Crassin[1], Pascal Guehl[2], Eric Heitz[2], Jérémy Jaussaud[2] et Fabrice Neyret[2]

[1]NVidia Research (France, Paris)
[2]Maverick, INRIA Grenoble-Rhône-Alpes et LJK (Université de Grenoble et CNRS)



Résumé

Nous présentons la librairie GigaVoxels utilisée pour rendre de très grandes scènes et objets détaillés en temps-réel. Implémentée en CUDA, elle tire parti des performances et fonctionnalités massivement parallèles des processeurs graphiques. Basée sur une représentation géométrique pré-filtrée volumique et associée à un algorithme de type "cone tracing" à base de voxels, elle permet un rendu haute performance avec une grande qualité de filtrage. La structure de données sous-jacente, un SVO (Sparse Voxel Octree), exploite le fait que dans les scènes 3D, les détails sont souvent concentrés sur leur interface et montre que les modèles volumétriques peuvent devenir une alternative intéressante en tant que primitive de rendu pour les applications temps-réel. Notre solution est basée sur une représentation hiérarchique adaptative de données en fonction du point de vue en cours, couplé à un algorithme de ray-casting. Le coeur du système, un mécanisme de cache implémenté GPU, offre une pagination des données en mémoire vidéo et est couplé à un pipeline de production de données capable de charger dynamiquement ou produire des voxels à la volée sur GPU. La production des données et la mise en cache dans la mémoire vidéo sont basées sur des requêtes de données et d'informations d'utilisation émises lors du rendu. Nous illustrons notre approche avec plusieurs applications.

Abstract

We present an implementation of the GigaVoxels rendering engine used to render large scenes and detailed objects in real-time. Implemented in CUDA, it leverages the performance and features of massively parallel graphics processors. It is based on a volumetric pre-filtered geometry representation and an associated voxel-based approximate cone tracing that allows a high performance rendering with high quality filtering. The underlying data structure, a SVO (Sparse Voxel Octree), exploits the fact that in CG scenes, details are often concentrated on their interface and shows that volumetric models might become a valuable alternative as a rendering primitive for real-time applications. Our solution is based on an adaptive hierarchical data representation depending on the current view, coupled to a ray-casting rendering algorithm. The core system, a GPU cache mechanism, provides a paging of data in video memory and is coupled with a data production pipeline able to dynamically load or produce voxel data on the GPU. Data production and caching in video memory is based on data requests and usage information emitted during rendering. We illustrate our approach with several applications.

Mots clé : Informatique graphique, rendu volumique, SVO, Octree, CUDA, GPU, cache

1. Introduction

La technologie "Gigavoxels" a été développée afin de permettre l'exploration en temps-réel et visuellement réaliste

d'immenses volumes détaillés, éventuellement créés à la volée. Cette technique vise un large type d'applications, depuis les jeux vidéos jusqu'aux effets spéciaux à la Digital Domain (`http://digitaldomain.com/`) (avalanches, fumée, nuages), en passant par la visualisation enrichie d'objets astrophysiques (projet ANR : galaxie, nébuleuses).

La phase de recherche exploratoire passée, nous en avons tiré une plateforme (moteur de rendu) robuste, utilisable par les chercheurs poursuivant les travaux sur le sujet, et pour les collaborations industrielles ou académiques. Un effort important a été porté sur la mise au point de l'API, du Devkit, des exemples et la qualité de la documentation. Nous poursuivons actuellement l'amélioration des performances et des fonctionnalités de base. L'API permet d'interfacer le rendu volumique avec un rendu OpenGL classique et de "cacher" le moteur, notamment la structure de données sous-jacente, le chargement dynamique des données ainsi que le système de cache. Elle donne également accès aux parties sur lesquelles les chercheurs expérimentent (équation de rendu du voxel, fabrication des briques de volume, stockage et interpolation de variables utilisateurs ajoutées aux voxels).

Dans ce qui suit, on illustre les fonctionnalités de GigaVoxels au travers d'exemples d'applications contenus dans notre kit de développement. En parallèle, nous présentons les travaux en cours et à venir.

2. Concepts clefs

GigaVoxels est adapté à des architectures limitées en mémoire et exploite les fonctionnalités des architectures massivement parallèles des GPU. Le but est de réutiliser les données autant que possible (système de cache). La structure de données interne unifie géométrie et texture. Elle est constituée d'une représentation multi-échelle pour laquelle le filtrage des données est bien défini. L'ajustement des niveaux de détails (LOD) est géré automatiquement. Cette structure est en outre pratique à traverser, éditer et efficace à rendre. Le rendu ne dépend que de ce qui est visible (ray-tracing). On ne charge que les données dont on a besoin, à la résolution voulue (LOD et streaming de données). Les voxels, quand à eux, permettent de gérer la transparence dûe à leur organisation spatiale, là où les triangles nécessitent des algorithmes de tris coûteux. Ils permettent également d'adresser les problèmes d'aliasing difficiles et coûteux à traiter pour des modèles à base de triangles.

2.1. Pipeline GigaVoxels

Un pipeline GigaVoxels regroupe les éléments suivants : une structure de données, un système de cache, un producer (production des voxels), un renderer et un shader (visualisation des voxels). Les deux éléments à définir par l'utilisateur sont le Producer et le Shader, l'API fournissant les mécanismes de base pour gérer les autres éléments.

2.2. Structure de données

La structure de données GigaVoxels est composée de deux éléments : une structure spatiale accélératrice de type N^3-Tree (en général un octree) et les voxels (c'est-à-dire les données utilisateurs). Les noeuds sont regroupés en tuile de noeuds et les voxels en briques de voxels. L'utilisateur définit d'une part la dimension de son N^3-Tree, en général $N = 2$ (octree), d'autre part la taille des bricks de voxels (ex : $8x8x8$). En outre, le contenu des voxels est définit via une liste de types (uchar4, float4, float, etc...)

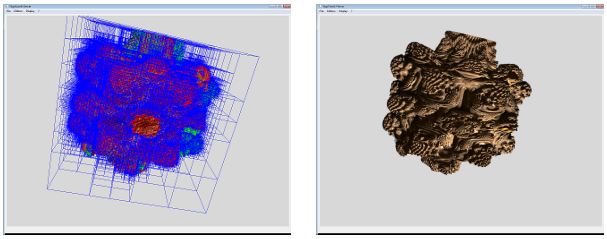


Figure 1: *Octree (gauche) et structure de données (droite) d'un Mandelbulb*

2.3. Système de cache

Conçu pour visualiser des données trop volumineuses pour tenir en mémoire, le coeur de GigaVoxels est un système de cache générique implémenté sur GPU. Son fonctionnement se base sur un mécanisme LRU (Least Recently Used) afin de pouvoir recycler les données les moins récentes. Chaque type de données contenu dans les voxels (couleur, normal, densité, etc...) est stocké sous forme d'une texture 3D accessible en lecture/écriture afin de bénéficier de l'interpolation hardware lorsque l'on souhaite lire (sampler) les données à une position particulière de l'espace.

2.4. Producer

Le producer produit les données utilisateurs (voxels). Attaché au cache, son but est de répondre à ses deux types de requêtes, soit : subdiviser un noeud de la structure spatiale (afin de pouvoir raffiner les données) ou produire/charger une brique de voxels (données utilisateurs). Il en existe deux types : procédural ou spécialisé dans le chargement de fichiers (streaming). Dans le cadre de la subdivision d'un noeud, à partir des positions spatiales des noeuds et voxels qui lui sont fournis, l'utilisateur doit déterminer si une région est vide ou si elle contient des données. Même chose pour la production d'une brique de voxels, le but est, soit de générer procéduralement les valeurs de chacun des types de données contenus dans un voxel, soit aller lire/charger le fichier de données associé à la brique.

2.5. Renderer et Shader

Lors de la phase de rendu par lancer de rayons, le Renderer traverse la structure spatiale et, suivant le contenu des noeuds, émet des requêtes de subdivision spatiale (raffinement) ou de production de briques de voxels. Une fois les voxels produits ou chargés en mémoire, le Shader a à disposition la position spatiale courante le long du rayon afin de pouvoir sampler une texture 3D contenant les données. Une fois récupérée, on peut par exemple effectuer un shading traditionnel comme Blinn/Phong. La normale peut soit

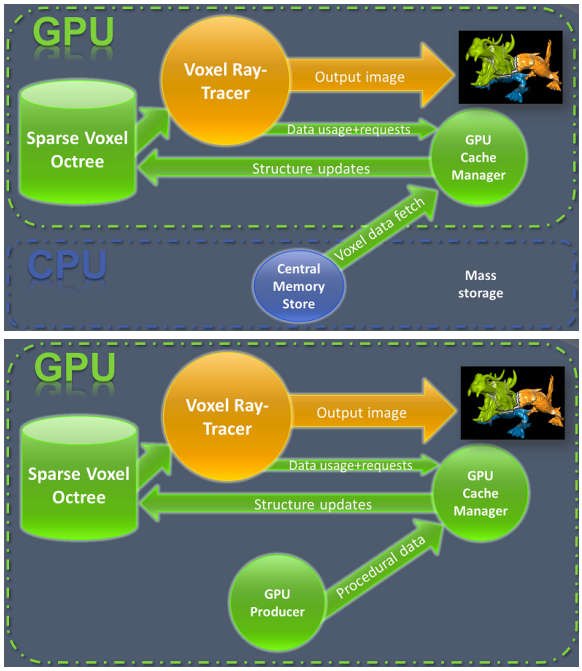


Figure 2: *Pipeline Gigavoxels : chargement de données (en haut), génération procédurale (en bas)*

être stockée dans les voxels, soit calculée dans le shader avec un gradient.

3. Chargement dynamique de données

Le producer est chargé de lire les données sur CPU (disque dur) et de les renvoyer sur GPU. Deux modes de chargement sont disponibles. On peut soit charger directement le niveau le plus résolu, soit charger successivement chaque niveau de détails du moins résolu au plus résolu.

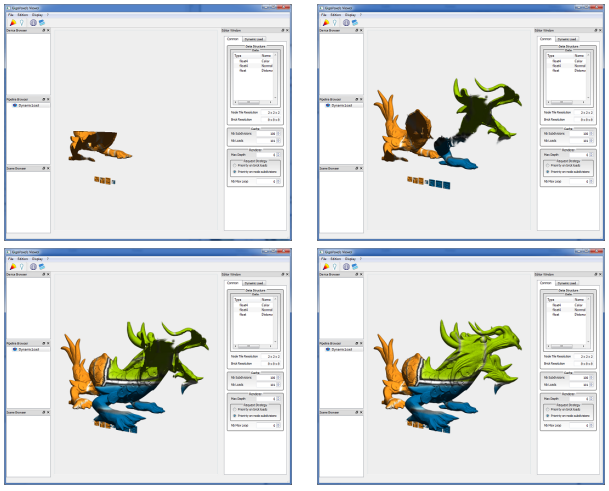


Figure 3: *Chargement de données à la résolution maximale (succession d'étapes)*

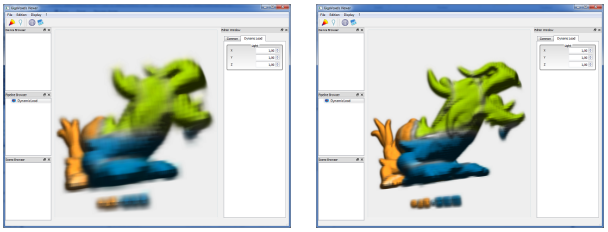


Figure 4: *Chargement de données progressif du moins résolu au plus résolu*

4. Importer vos données

L'API fournit des mécanismes de base pour voxéliser vos données. Dans l'exemple ci-dessous, un fichier traditionnellement utilisé en visualisation scientifique est voxelisé au chargement puis visualisé avec le Producer de "chargement dynamique de données" vu précédemment. Un seuillage est appliqué pour enlever du bruit.

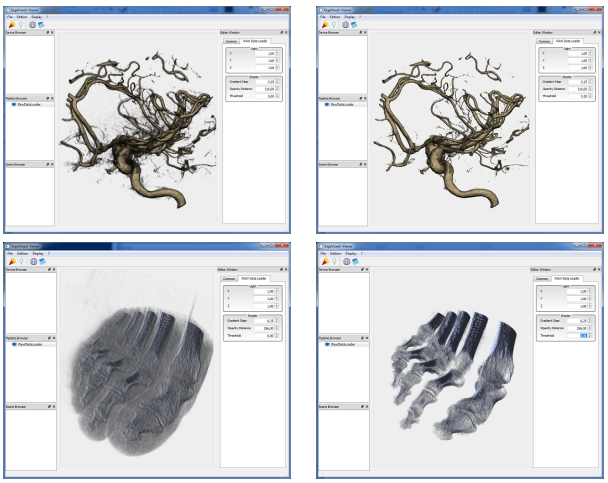


Figure 5: *Modèles d'anévrisme et de pied importés depuis des fichiers RAW*

Nous travaillons actuellement à l'ajout de fonctionnalités de clipping plane afin, par exemple, de pouvoir progresser dans un volume en ne sélectionnant qu'une région d'intérêt.

5. Génération procédurale de données

La génération procédurale de données permet d'ajouter des détails à une scène et, dans le cas d'objets de type fractal, permet de générer des environnements 3D tout en réduisant à la consommation mémoire.

Dans l'exemple qui suit, on génère procéduralement sur GPU un objet de type fractal appelé Mandelbulb, un analogue en dimension 3 de l'ensemble de Mandelbrot. Il est défini par :

$$\langle x,y,z\rangle^n = r^n\langle \cos(n\theta)\cos(n\phi), \sin(n\theta)\cos(n\phi), \sin(n\phi)\rangle$$

$$\text{où} \left\{ \begin{array}{rcl} r & = & \sqrt{x^2+y^2+z^2} \\ \theta & = & \arctan(y/x) \\ \phi & = & \arctan(z/\sqrt{x^2+y^2}) = \arcsin(z/r) \end{array} \right\}$$

pour la nième puissance du nombre hypercomplexe 3D. Les points sont calculés par itération de $z \mapsto z^n + c$ où z et c sont des nombres hypercomplexes dans un espace de dimension 3 et $z \mapsto z^n$ l'application définie ci-dessus. Ici $n = 8$.

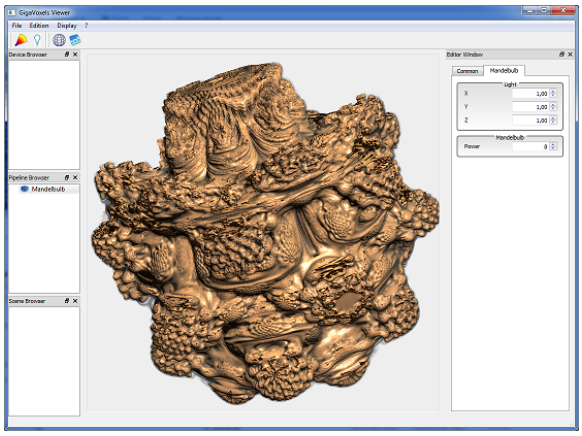


Figure 6: *Génération procédurale d'un Mandelbulb sur GPU*

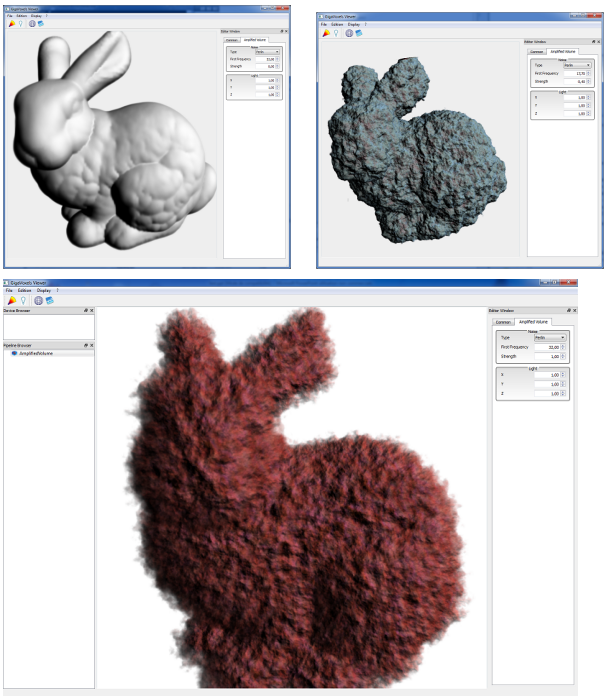


Figure 7: *Modèle de lapin (haut gauche) changé en pierre via un bruit de Perlin (haut) et habillé de fourrure (en bas, en jouant avec la fonction de transfert)*

6. Ajout de détails : bruit

Le but envisagé est de pouvoir ajouter du détail à des objets.

Dans cet exemple, on souhaite ajouter du détail à un modèle 3D de Bunny représenté sous la forme d'un Signed Distance Field (distance field signé), c'est-à-dire qu'il contient une normale et une distance. On perturbe la distance et la normale en appliquant un bruit de Perlin.

La structure de données GigaVoxels est constituée d'un octree et de voxels contenant deux éléments, une couleur RGBA et une normale. Le Producer récupère le modèle 3D du Bunny via une texture3D. La couleur est calculée dans chaque voxel comme suit : on calcule un bruit de Perlin à la position courante (centre du voxel courant), on récupère la distance du Bunny en ce point (sample la texture 3D), on applique cette perturbation sur la distance, puis on applique une fonction de transfert sur cette nouvelle distance afin d'obtenir une couleur RGBA. Même chose pour la normale, sans application d'une fonction de transfert. Le Shader sample les couleurs et les normales calculées par le Producer en appliquant un shading diffus.

L'avantage d'une représentation volumique est de pallier aux problèmes de parallax que l'on retrouve par exemple dans le Bump-Mapping.

Nous travaillons actuellement sur des optimisations de code afin de ne pas recalculer le bruit à chaque étape mais plutôt stocker le bruit, une octave par une, dans le cache sur GPU. Ainsi, à chaque niveau de résolution, on ajoute une octave qui a déjà été calculée précédemment.

7. Ajout de détails : hypertextures

Une hypertexture est une texture 3D à laquelle on a ajouté une notion d'épaisseur dans laquelle on va pouvoir ajouter du détail. Par exemple une côte de maille d'un personnage 3D.

On définit pour n'importe quel objet une fonction caractéristique qui définit un mapping de $f : \mathbb{R}^3 \longrightarrow [0;1]$. Tous les points $\vec{x}$ pour lesquels $f(\vec{x})$ est égale à 0 sont considérés comme étant en dehors de l'objet. Tous les points $\vec{x}$ pour lesquels $f(\vec{x})$ est égale à 1 sont dits "étant strictement à l'intérieur de l'objet". Finalement, tous les points $\vec{x}$ pour lesquels $0 < f(\vec{x}) < 1$ sont considérés comme étant dans une région "fuzzy". Cette formulation donne à un objet surfacique une épaisseur dans laquelle on va avoir du détail.

On considère une fonction de distance $D(\vec{x})$ à une surface proxy simple où l'on veut afficher de l'hypertexture sous la peau sur une épaisseur E. Par exemple, $D(\vec{x}) = \|\vec{x}\| - rayon$ pour une sphère. On utilise également une fonction pour seuiller/lisser les résultats,

$$\chi(x) = smoothStep(0.5 - \varepsilon, 0.5 + \varepsilon)$$

La densité en un point $\vec{x}$ est alors donnée par la relation :

$$density(\vec{x}) = \chi(-D(\vec{x})/E + a.noise(b.\vec{x}))$$

Les hypertextures sont très coûteuses à calculer. Nous travaillons actuellement sur des optimisations de code. Mathématiquement, il est possible de prédire si l'ajout d'une octave de bruit va nous amener dans une zone en dehors de l'épaisseur d'une hypertexture, ce qui permet de stopper les calculs prématurément.

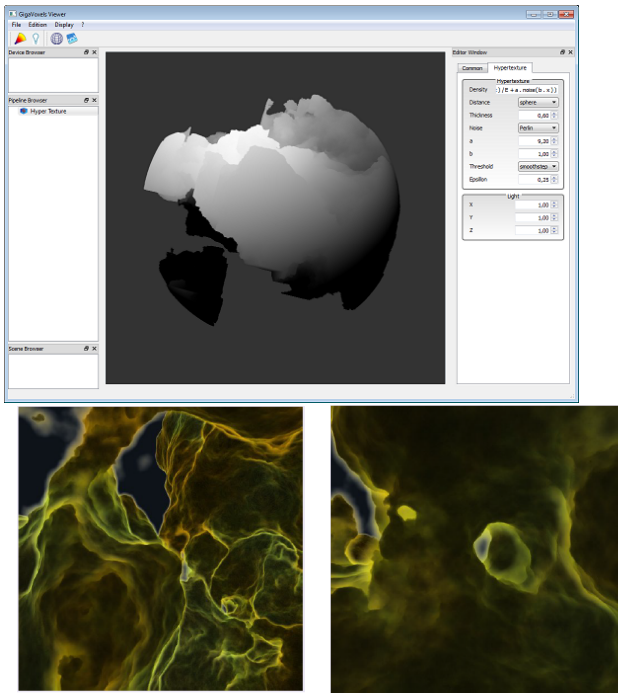


Figure 8: *Hypertexture sur une sphère (haut), à l'intérieure de l'hypertexture colorée (bas)*

8. Depth of Field

Afin d'ajouter plus de réalisme aux images produites, l'API du Shader permet de simuler l'action d'une lentille de caméra générant un flou de profondeur.

Il s'agit de modifier l'ouverture du cône utilisé lors du rendu par "Cone Tracing". On obtient alors un double cône à l'intérieur duquel les niveaux de résolutions sélectionnés dépendent de la taille du cercle de confusion. Contrairement aux scènes en triangles où le depth of field est un processus coûteux à réaliser, pour la technologie GigaVoxels, plus il y a de flou, plus la scène est rapide à rendre et moins il y a de mémoire nécessaire pour représenter la scène.

9. Synthèse de données

Afin de pouvoir générer des environnements 3D, la structure de données GigaVoxels permet de réaliser de l'instanciation de branches internes du graphe ainsi que de la récursivité grâce à un système de pointeur de noeuds. Cela peut notamment s'appliquer sur des structures fractales et réduit la consommation mémoire de façon significative.

On montre ici un exemple de structure spatiale N^3-tree pour laquelle $N = 3$. C'est un objet de type fractal appelé éponge de Menger/Serpinski. Une seule brique de voxels est produite dans le premier noeud. Ensuite, tous les autres noeuds de la structure spatiale pointent sur lui.

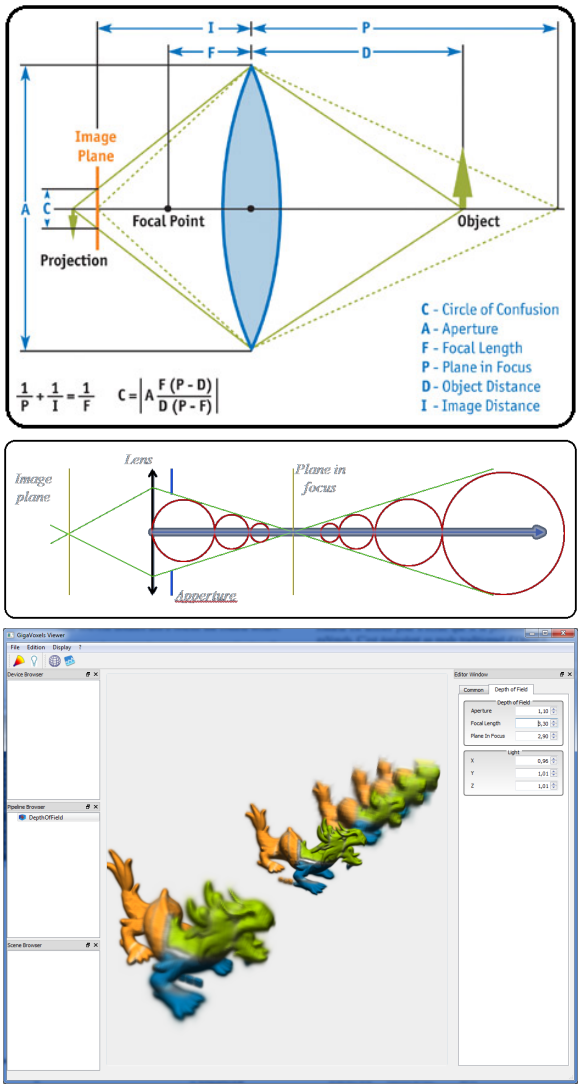


Figure 9: *Depth of Field : schéma physique (haut), double cône de la lentille GigaVoxels (milieu, les cercles rouges indiquent les différents niveaux de résolution utilisés) et rendu final (bas)*

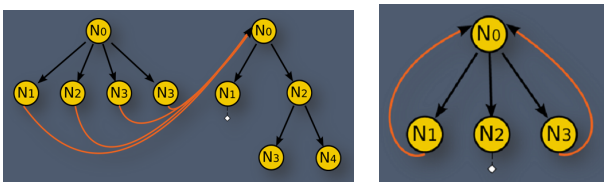


Figure 10: *Synthèse de données : instanciation (gauche) et récursivité (droite)*

10. Interopérabilité OpenGL

L'API GigaVoxels fournit des mécanismes de base nécessaires permettant de mélanger une scène en triangles classique avec des voxels.

L'API CUDA fournit pour cela des ressources graphiques

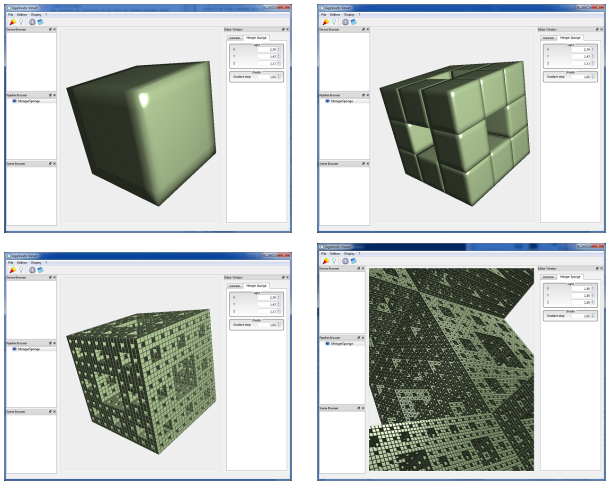


Figure 11: *Récursivité du graphe sur une éponge de Menger*

mappant des buffers OpenGL dans l'espace mémoire de CUDA. Ces objets sont alors directement accessibles en lecture/écriture dans des programmes CUDA. On récupère notamment les informations de couleurs et profondeur du FrameBuffer OpenGL. Actuellement, les mécanismes mis en oeuvre permettent de récupérer : PBO (Pixel Buffer Object), Texture Object et RBO (Render Buffer Object).

Ci-dessous, un exemple de jeu vidéo basique. Des vaisseaux spatiaux OpenGL évoluent dans une structure GigaVoxels sous forme d'éponge de Menger/Serpenski (vu précédemment) et tirent des missiles générant des explosions. Des exemples plus complets et orientés jeux vidéos vont être développés par la suite.

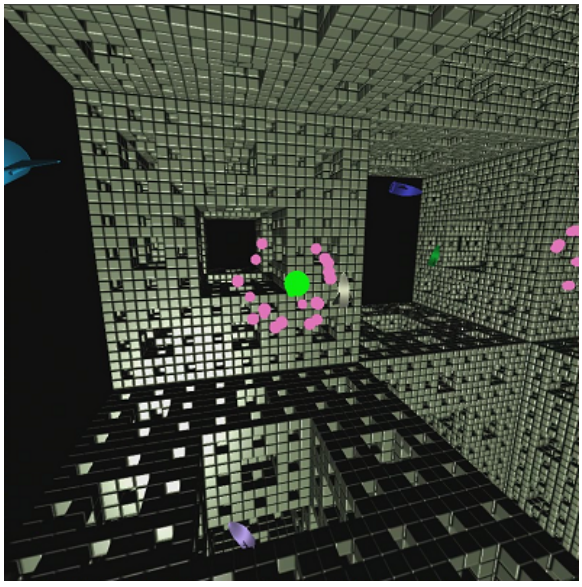


Figure 12: *Vaisseaux spatiaux OpenGL dans une structure GigaVoxels de type fractale (éponge de Menger Serpinski)*

11. Instanciation d'objets

Toujours dans l'idée de pouvoir générer des environnements 3D complexes, nous travaillons sur l'ajout de fonctionnalités pour gérer plusieurs objets dans une scène.

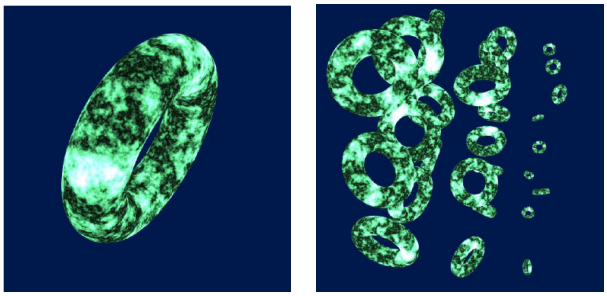


Figure 13: *Instanciation d'objets : tore (gauche) et instanciation de plusieurs tores (droite)*

Le but serait de permettre au système de cache de gérer plusieurs N^3-tree. Différentes options peuvent être envisagées. Par exemple, lors du rendu d'une forêt, on souhaite fournir une collection de zones 3D (proxy geometry) dans lesquelles afficher des arbres sous forme volumique.

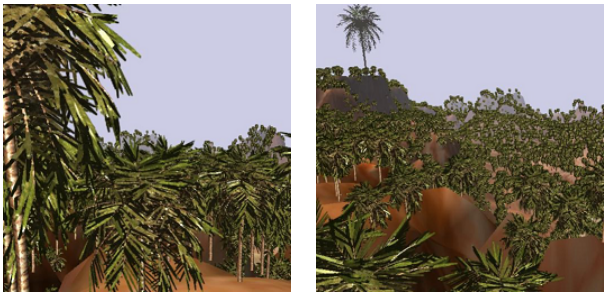


Figure 14: *But encore à l'étude : instanciation de plusieurs arbres*

On peut envisager diverses solutions, à savoir : récupérer la boîte englobante 2D d'un objet projeté à l'écran, rastériser la bounding box GigaVoxels en générant les rayons en chacun des pixels depuis la caméra, puis renvoyer ce buffer à GigaVoxels via l'interopérabilité CUDA/OpenGL et un fragment shader GLSL. Ceci afin de bénéficier de la rastérization hardware. En outre, Si plusieurs objets sont présents dans la scène, il serait intéressant d'étudier les coûts des transferts mémoires entre OpenGL et la mémoire CUDA GigaVoxels et de récupérer chacune des proxy géometries ou les clusteriser dans une seule, voire plusieurs, quitte à raffiner d'avantages certaines zones.

12. Projet ANR : RTIGE

Nous travaillons dans le cadre du projet ANR "RTIGE" (Real-Time & Interactive Galaxy for Edutainment) afin de visualiser des modèles de galaxies et de nébuleuses.

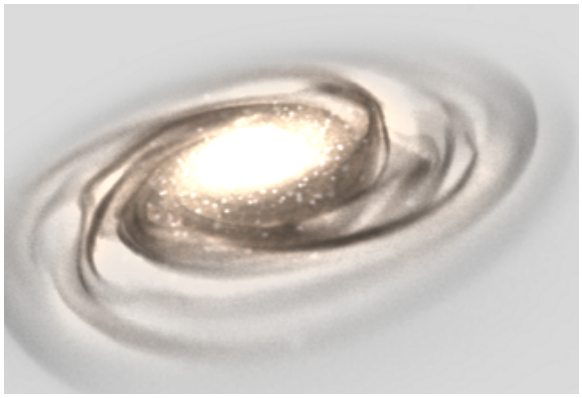


Figure 15: *Projet ANR RTIGE : visualisation de galaxies*

13. Outils de développement

Afin de pouvoir tester, valider nos recherches et fournir un environnement d'étude évolutif, le kit de développement à été étendu afin de fournir des outils facilitant le développement.

13.1. Voxeliseur [génération de données]

On fournit un outil pour voxeliser une scène 3D au format GigaVoxels. Cette première version permet de générer des données de type "unsigned char" à 4 composantes (ex : de type RGBA). Un seul canal est disponible. La voxelisation est effectuée en pré-traitement. Le résultat produit des fichiers de noeuds et des fichiers de bricks qui pourront être chargés à la volée par un "producer" GigaVoxels.

Cet outil est en cours d'amélioration. Notamment pour gérer plus de types de données et de pourvoir agir en temps-réel. Certaines avancés permettent désormais de voxéliser des scènes à la volée (crasin, Décoret, Eisemann, etc...).

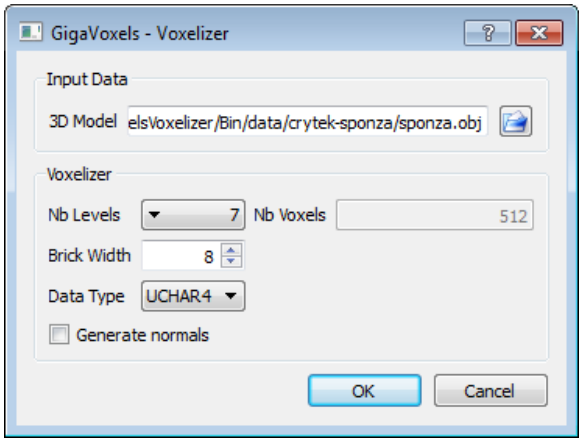


Figure 16: *Voxeliseur GigaVoxels*

13.2. Viewer [environnement d'études]

Afin de permettre de se servir de GigaVoxels comme d'un outil d'études, un viewer a été développé avec des éditeurs associés à chaque élément GigaVoxels (un moyen de pallier le problème de la recompilation à la volée comme dans le langage de shader programmable GLSL). Les paramètres éditables peuvent être des constantes CUDA équivalents des Uniforms GLSL exécutées dans le code GPU. L'arrivée de CUDA 5 et sa fonctionnalité de compilation séparée devrait grandement simplifier la programmation de GigaVoxels.

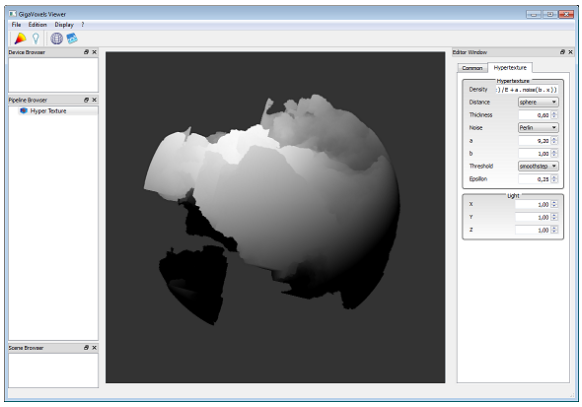


Figure 17: *Viewer GigaVoxels*

13.3. Analyse de performances et outils de débogage

GigaVoxels fournit une API permettant de mesurer le temps exécuté entre deux évènements sur CPU et GPU. Voici un exemple d'utilisation du Performnance Monitor sur le modèle de lapin bruité vu précédemment :

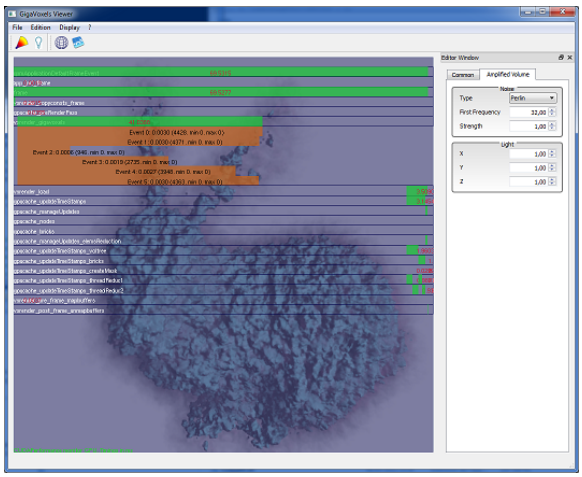


Figure 18: *L'outil Performnace Monitor sur un lapin bruité*

Nous sommes actuellement en phase de profilage et étude de l'amélioration des performances. Nous travaillons à l'ajout de fonctionnalités pour établir des métriques (nombres de briques produites par seconde, etc...), budget de temps à respecter pour rendre une frame, gestion de priorité sur des briques, etc...

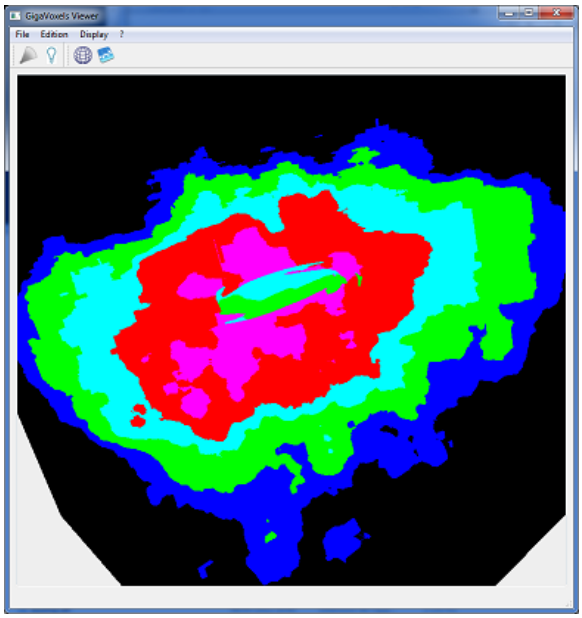


Figure 19: *Exemple d'estimation du coût par rayon sur un modèle de galaxie (forme de disque entouré d'un nuage de poussière)*

14. Informations utiles

Un site dédié à GigaVoxels regroupant publications, galerie d'images, vidéos, code source et documentation se trouve à l'adresse suivante : `http://gigavoxels.inrialpes.fr/`

GigaVoxels est distribué sous un schéma double licence. Vous pouvez obtenir une licence spécifique de l'INRIA à l'adresse gigavoxels-licensing@inria.fr. Autrement, la licence par défaut est la GPL version 3.

La librairie est écrite en langage C++ et fait une utilisation massive des templates. Elle est basée sur l'API CUDA 4.2 et nécessite des processeurs graphiques de Compute Capability supérieure ou égale à 2.0.

15. Remerciements

GigaVoxels est soutenu et financé dans le cadre du projet ANR "RTIGE" (Real-Time & Interactive Galaxy for Edutainment) de référence est ANR-10-CORD-0006.

16. Conclusion

Faisant suite à une phase de recherche exploratoire, une première version officielle de la technologie "Gigavoxels" vient d'être créée.

Cette librairie a été développée afin de permettre l'exploration en temps-réel et visuellement réaliste d'immenses volumes détaillés. Elle est disponible pour les chercheurs poursuivant les travaux sur le sujet, et pour les collaborations industrielles ou académiques. Elle offre des outils, un environnement d'études et des tutoriaux documentés détaillant la marche à suivre afin de réaliser des fonctionnalités telles que : le chargement dynamique de données, l'ajout de détails générés procéduralement via du bruit et des hypertextures, le depth of field, la synthèse de données par instanciation et la récursion de données de la structure de données, etc...

Certaines fonctionnalités sont en cours de développement, telles que l'instanciation d'objets pour visualiser plusieurs objets (à la manière du geometry instancing), l'interopérabilité avec la librairie OpenGL afin de mixer des scènes en voxels et en triangles.

Un effort important a été porté sur la mise au point de l'API, du Devkit, des exemples ainsi que la qualité de la documentation. Nous poursuivons actuellement des efforts sur l'amélioration des performances.

Il est également possible de se servir de GigaVoxels pour d'autres choses, comme l'illumination globale. Nous réfléchissons également à la possibilité de faire de l'animation via le skinning de shellmaps.

Références

[Car08] CARMACK J. : John carmack on id tech 6, ray tracing, consoles, physics and more. id Software.

[CNSE10] CRASSIN C., NEYRET F., SAINZ M., EISEMANN E. : *Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In book : GPU Pro*. A K Peters, 2010, ch. X.3, pp. 643–676.

[Cra11] CRASSIN C. : *GigaVoxels : A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. English and web-optimized version.

[Oli08] OLICK J. : Current generation parallelism in games. id Software.

[Per03] PERLIN K. : Noise, hypertexture, antialiasing, and gesture. In *Texture and Modeling, A Procedural Approach (Third Edition)*. 2003, ch. 12, pp. 337–410.